ElasticSearch

ElasticSearch概述

简称es，是一个开源的高扩展的分布式全文搜索引擎。近乎实时的存储、检索数据，扩展性很强。使用Java进行开发并使用Lucene作为其核心来实现所有索引和搜索的功能，其目的是通过简单的RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。其用于全文搜索，结构化搜索，分析以及三者混用。

Lucene是一个开放源代码的全文检索引擎工具包，但不是一个完整的全文检索引擎，而相当于一个架构。当前是最受欢迎的免费java信息检索程序库。

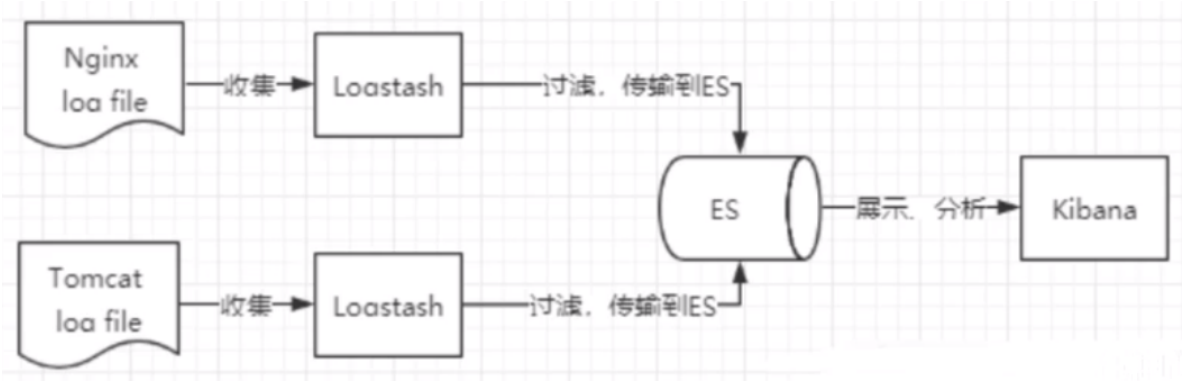
Elasticsearch 是一个分布式的开源搜索和分析引擎，适用于所有类型的数据，包括文本、数字、地理空间、结构化和非结构化数据。Elasticsearch 在 Apache Lucene 的基础上开发而成，由 Elasticsearch N.V.（即现在的 Elastic）于 2010 年首次发布。Elasticsearch 以其简单的 REST 风格 API、分布式特性、速度和可扩展性而闻名，是 Elastic Stack 的核心组件；Elastic Stack 是适用于数据采集、充实、存储、分析和可视化的一组开源工具。人们通常将 Elastic Stack 称为 ELK Stack（代指 Elasticsearch、Logstash 和 Kibana），目前 Elastic Stack 包括一系列丰富的轻量型数据采集代理，这些代理统称为 Beats，可用来向 Elasticsearch 发送数据。

ES使用场景：

- 维基百科，全文检索，高亮，搜索推荐
- 新闻网站，用户行为日志，社交网络数据，文章用户反馈
- Stack Overflow，GitHub
- 电商网站，检索商品
- 日志数据分析，logstash采集日志，复杂数据分析，ELK技术，ElasticSearch+logstash+kibana
- 商品价格监控网站
- BI系统，商业智能
- ……

ES和Solr差别：

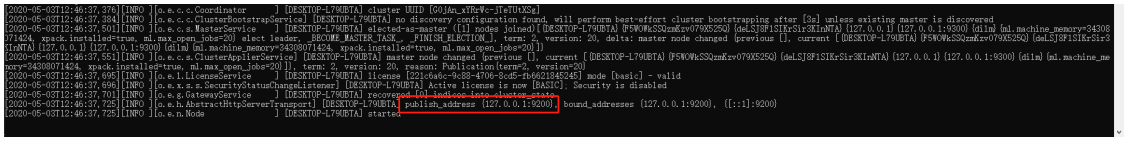
1. 单纯地对已有数据进行搜索时Solr更快
2. 当实时建立索引时，Solr会产生IO阻塞，查询性能较差，ES具有明显的优势（尤其是大数据时代）
3. Solr使用Zookeeper进行分布式管理，而ES自身带有分布式协调管理功能，更方便
4. Solr支持更多的格式，ES仅支持json



安装配置

ElasticSearch安装

需要保证ES版本和Java版本对应。

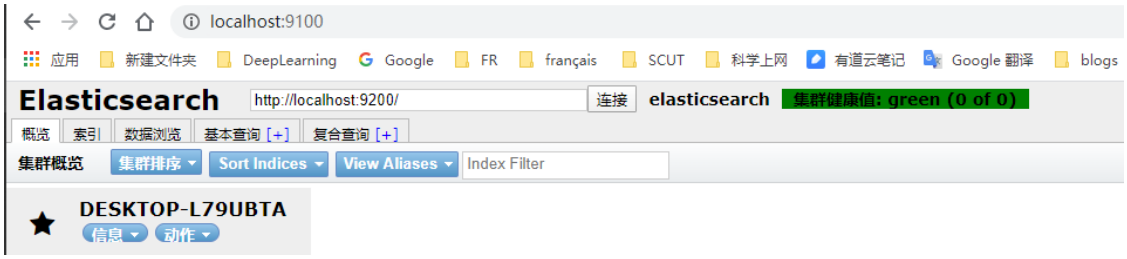
1. 下载ES：https://www.elastic.co/cn/
2. 解压
3. 在bin文件夹下启动`elasticsearch.bat`文件
4. 就会发现这样提示：可以访问127.0.0.1:9200
5. 
6. 浏览器会有这样的信息：
7. ```
   {
     "name" : "DESKTOP-L79UBTA",
     "cluster_name" : "elasticsearch",
     "cluster_uuid" : "GOjAn_xYRrWc-jTeTUtXSg",
     "version" : {
       "number" : "7.6.2",
       "build_flavor" : "default",
       "build_type" : "zip",
       "build_hash" : "ef48eb35cf30adf4db14086e8aabd07ef6fb113f",
       "build_date" : "2020-03-26T06:34:37.794943Z",
       "build_snapshot" : false,
       "lucene_version" : "8.4.0",
       "minimum_wire_compatibility_version" : "6.8.0",
       "minimum_index_compatibility_version" : "6.0.0-beta1"
     },
     "tagline" : "You Know, for Search"
   }
   ```

安装可视化界面ES-head

1. 下载https://github.com/mobz/elasticsearch-head，解压
2. 如果没有node.js还要下载并安装https://nodejs.org/en/download/
3. 在目录下：
   1. `npm install`
   2. `npm run start`
4. 连接测试会发现存在跨域问题：需要关闭ES，并在`./bin/elasticsearch.yml`进行配置：

   ```
   http.cors.enabled: true
   http.cors.allow-origin: "*"
   ```

5. 重启ES服务再次连接http://localhost:9100/：

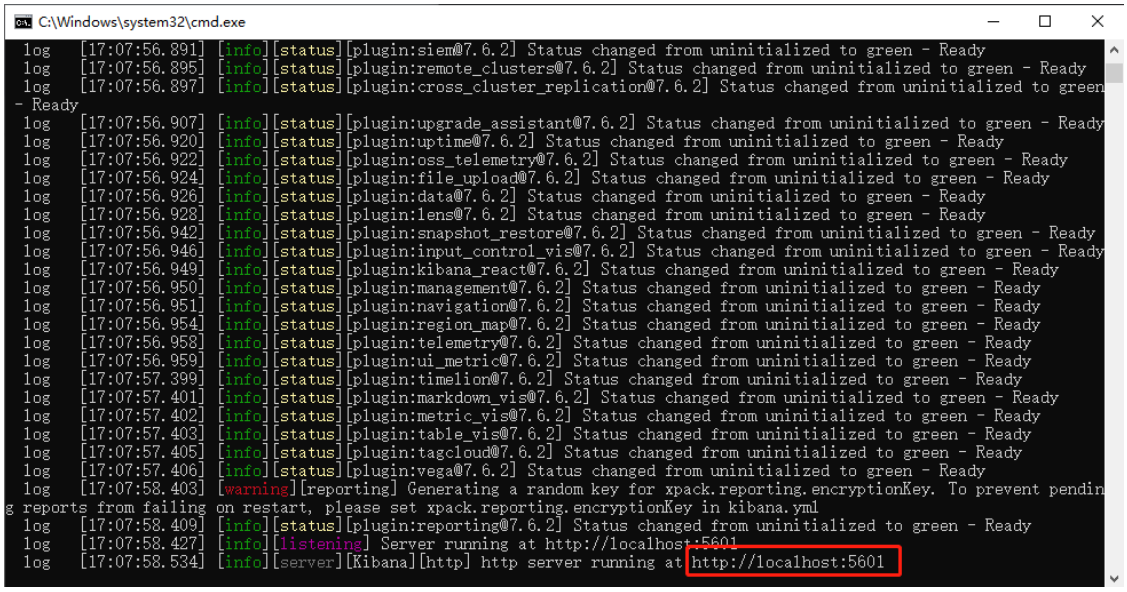


目前可以把ES当成一个数据库，索引当做库，文档当成库中的数据

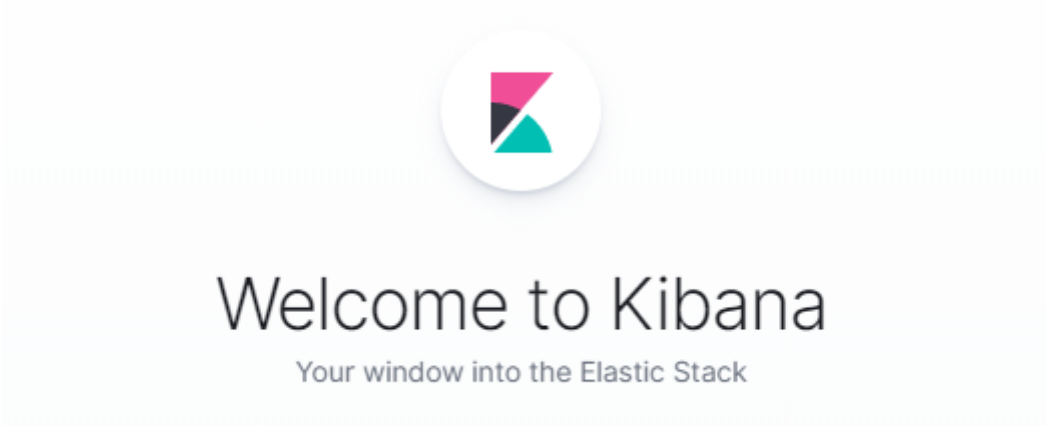
Kibana安装

1. 下载https://www.elastic.co/cn/downloads/kibana，解压
   ```

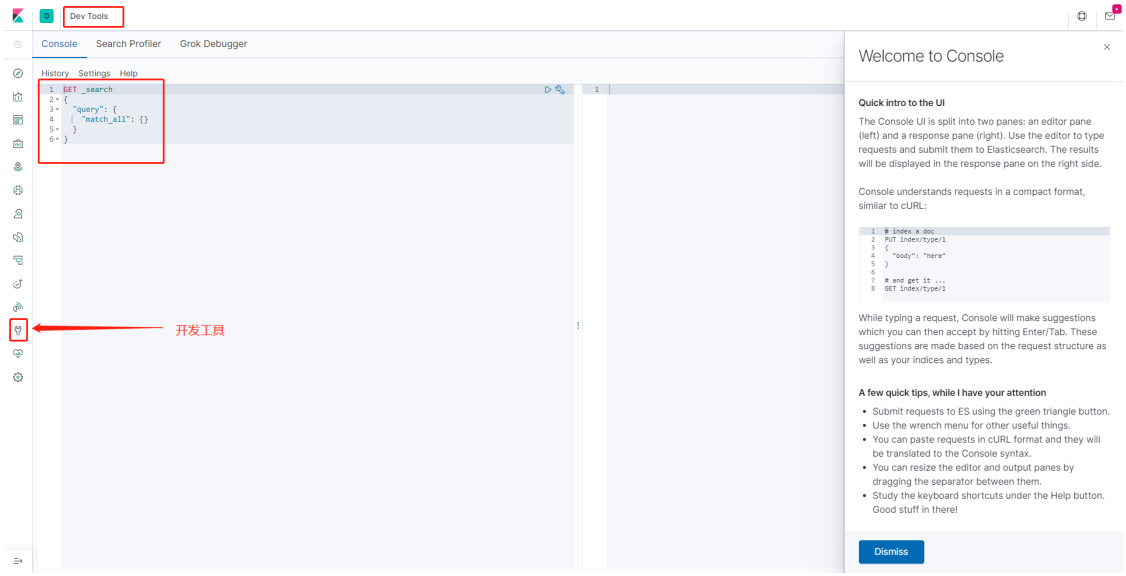
2. 在bin目录下运行`kibana.bat`文件：



3. 打开http://localhost:5601：



4. 把kibana当做开发工具：



5. 汉化：在`config/kibana.yml`中修改↓，并重启应用

```
i18n.locale: "zh-CN"
```

ElasticSearch核心概念：

> ElasticSearch是**面向文档**的

Relational DB	ElasticSearch
数据库 database	索引 indices
表 tables	types (逐渐弃用)
行 rows	documents
字段 columns	fields

物理设计：ElasticSearch在后台把**每个索引划分为多个分片**，每个分片可以在集群中的不同服务器间迁移

文档documents：

相当于数据库中的一条条数据。索引和搜索数据的最小单位就是文档。

具有几个重要属性：

- 自我包含，一篇文档同时包含字段和对应的值，即key-value
- 可以是层次型的，一个文档中包含自文档
- 灵活的结构，文档不依赖预先定义的模式

索引：

就是数据库。是映射类型的容器，即一个非常大的文档集合。索引存储了映射类型的字段和其他设置，然后他们被存储到了各个分片上。

倒排索引：

ElasticSearch使用到一种称为倒排索引的结构，采用Lucene倒排索引作为底层，这种结构**适用于快速的全文搜索**，一个索引由文档中所有不重复的列表构成，对于每一个词，都有一个包含他的文档列表。

为了创建倒排索引，我们首先要将每个文档拆分成独立的词（词条，tokens），然后创建一个包含所有不重复的词条的排序列表，然后列出每个词条出现在哪个文档里：

比如我们有

- 文档1：Study every day, good good up to forever
- 文档2：To forever, study every day, good good up

Term	文档1	文档2
Study	√	×
To	×	√
every	√	√
forever	√	√
day	√	√
study	×	√
good	√	√
to	√	×
up	√	√

这时如果我们尝试搜索 to forever，只需要查看包含每个词条的文档：

Term	文档1	文档2
to	√	×
forever	√	√
总计	2	1

可以看出文档1的匹配程度更高，所以在没有别的条件情况下认为文档1的权重更高。

再来一个例子，比如我们通过博客标签来搜索博客文章：

博客文章 原始数据	博客文章 原始数据	索引列表 倒排索引	索引列表 倒排索引
博客ID	标签	标签	博客ID
1	python	python	1,2,3
2	python	Linux	3,4
3	Linux，python		
4	Linux		

如果要搜索含有python标签的文章，则查找倒排索引后的数据将会快的多，只需要查看标签这一栏然后获取相关的文章ID即可。

在ElasticSearch中，索引被分为多个分片，每个分片是一个Lucene的索引，所以说**一个ElasticSearch索引是由多个Lucene索引组成的。**

IK分词器插件

下载https://github.com/medcl/elasticsearch-analysis-ik中找到的https://github.com/medcl/elasticsearch-analysis-ik/releases对应zip文件，解压，放在ElasticSearch的plugins文件夹中，重启ES和kibana服务。

使用kibana进行开发：

这里测试对一串中文进行最细粒度分词：

```
GET _analyze
{
  "analyzer": "ik_max_word",
  "text": "中华人民共和国"
}
```


```
{
  "tokens" : [
    {
      "token" : "中华人民共和国",
      "start_offset" : 0,
      "end_offset" : 7,
      "type" : "CN_WORD",
      "position" : 0
    },
    {
      "token" : "中华人民",
      "start_offset" : 0,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 1
    },
    {
      "token" : "中华",
      "start_offset" : 0,
      "end_offset" : 2,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
      "token" : "华人",
      "start_offset" : 1,
      "end_offset" : 3,
      "type" : "CN_WORD",
      "position" : 3
    },
    {
      "token" : "人民共和国",
      "start_offset" : 2,
      "end_offset" : 7,
      "type" : "CN_WORD",
      "position" : 4
    },
    {
      "token" : "人民",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 5
    },
    {
      "token" : "共和国",
```


这里测试了自动分词：

```

GET _analyze
{
  "analyzer": "ik_smart",
  "text": "中华人民共和国"
}
```


```
{
  "tokens" : [
    {
      "token" : "中华人民共和国",
      "start_offset" : 0,
      "end_offset" : 7,
      "type" : "CN_WORD",
      "position" : 0
    }
  ]
}

```


对于那些不在默认字典中的词，我们可以**自行添加字典**：在`ES\plugins\ik\IKAnalyzer.cfg`中我们找到：

```
<properties>
    <comment>IK Analyzer 扩展配置</comment>
    <!--用户可以在这里配置自己的扩展字典 -->
    <entry key="ext_dict">yulin.dic</entry>
     <!--用户可以在这里配置自己的扩展停止词字典-->
    <entry key="ext_stopwords"></entry>
    <!--用户可以在这里配置远程扩展字典 -->
    <!-- <entry key="remote_ext_dict">words_location</entry> -->
    <!--用户可以在这里配置远程扩展停止词字典-->
    <!-- <entry key="remote_ext_stopwords">words_location</entry> -->
```

```
</properties>    <comment>IK Analyzer 扩展配置</comment>
    <!--用户可以在这里配置自己的扩展字典 -->
    <entry key="ext_dict"/>
    <!--用户可以在这里配置自己的扩展停止词字典-->
    <entry key="ext_stopwords"/>
    <!--用户可以在这里配置远程扩展字典 -->
    <!-- <entry key="remote_ext_dict">words_location</entry> -->
    <!--用户可以在这里配置远程扩展停止词字典-->
    <!-- <entry key="remote_ext_stopwords">words_location</entry> -->
</properties>
```

同级目录下也发现了很多的字典文件`.dic`，因此我们可以模仿构建`yulin.dic`并如上进行注入。

注意要**重启ES**才能使得新增的词典生效，即使是修改一下词典的内容。

Rest风格操作

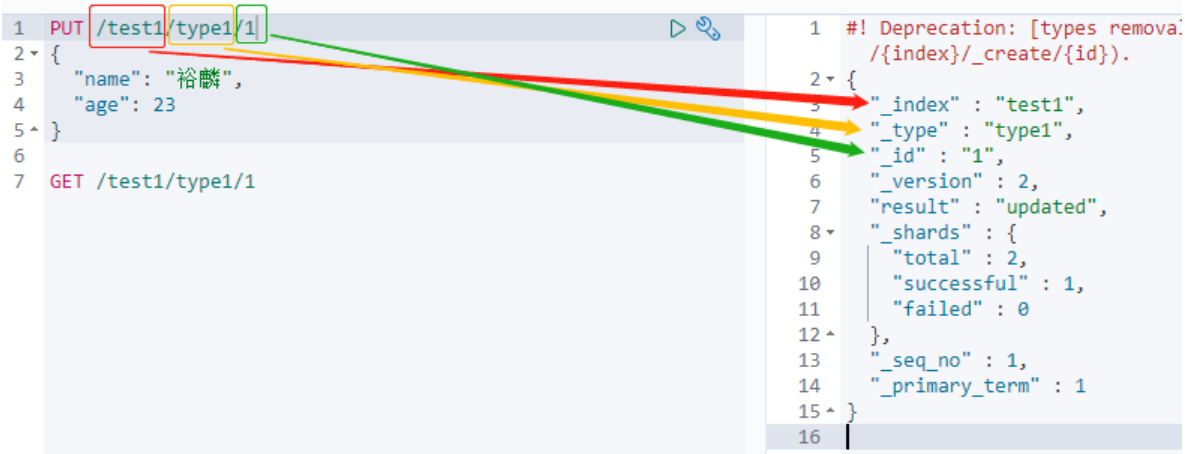
Rest风格是一种软件架构风格，而不是标准，只是提供了一组设计原则和约束条件。其主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

方法	URL地址	描述
PUT	localhost:9200/索引名称/类型名称/文档id	创建文档（指定文档id）
POST	localhost:9200/索引名称/类型名称	创建文档（随机文档id）
POST	localhost:9200/索引名称/类型名称/文档id/_update	修改文档
DELETE	localhost:9200/索引名称/类型名称/文档id	删除文档
GET	localhost:9200/索引名称/类型名称/文档id	通过文档id查询文档
POST	localhost:9200/索引名称/类型名称/文档id/_search	查询所有数据

索引的操作：

创建一个索引：

这里我在kibana添加了两次同一个文档，所以出现了`_version: 2`和`result: updated`的情况：



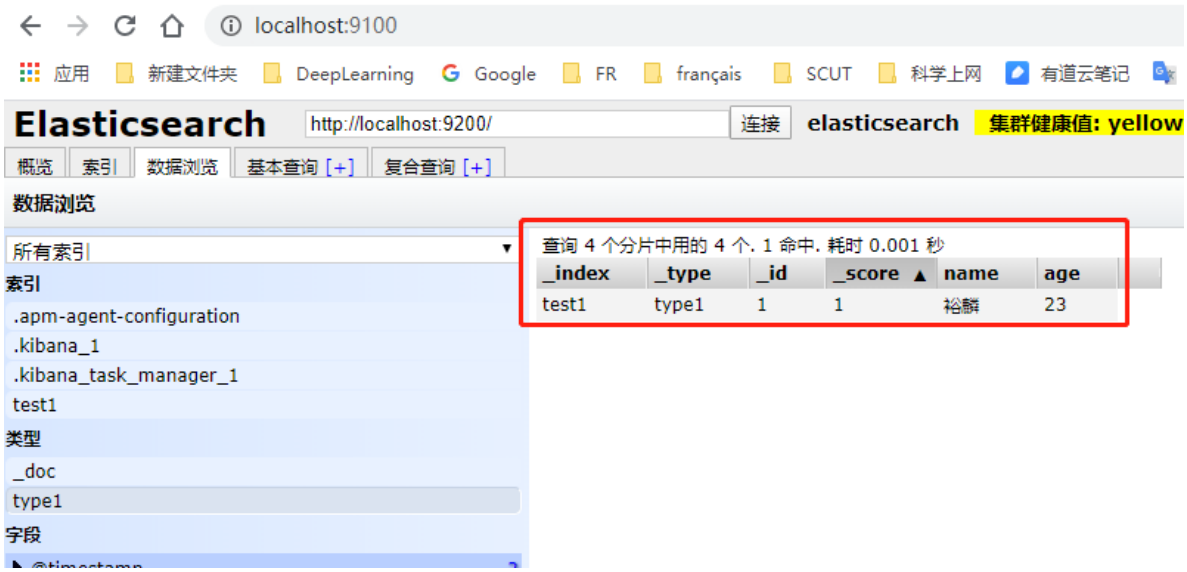
使用GET请求也能获取到数据：

```
PUT /test1/type1/1
{
  "name": "裕麟",
  "age": 23
}

GET /test1/type1/1
```

```
#! Deprecation: [types r
{
  "_index" : "test1",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 2,
  "_seq_no" : 1,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "裕麟",
    "age" : 23
  }
}
```

同样地我们也可以用ES-head可视化工具来看到数据：

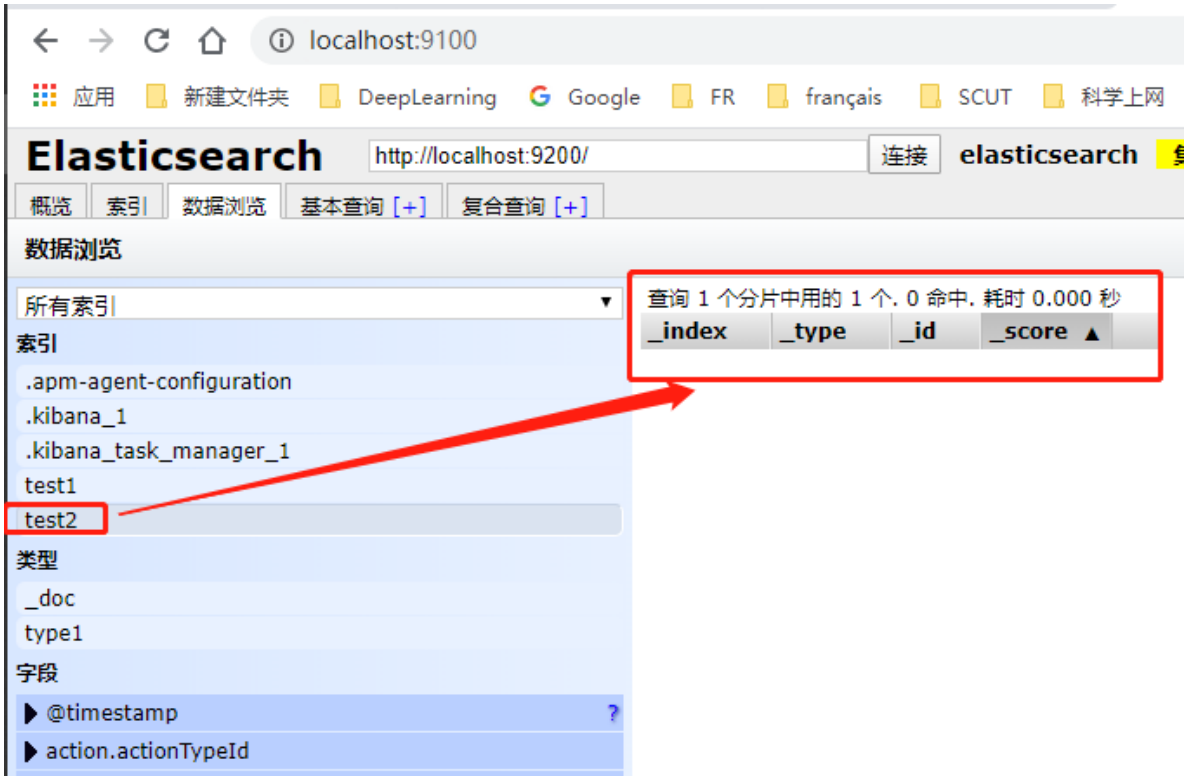


我们也可以只进行**创建索引而不插入文档**：

```
PUT /test2
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "age": {
        "type": "integer"
      },
      "birthday": {
        "type": "date"
      }
    }
  }
}
```

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "test2"
}
```

在ES-head中也可以看到该索引中并没有文档：



其实现在我们一般**使用 `_doc` 来表示默认类型**，在这种情况下ES会自动识别并配置字段类型：

扩展：使用`GET _cat/....`可以获取ES的很多当前信息

修改文档信息：可以使用PUT直接改，但是会有遗漏字段导致重新生成索引的风险，可以使用下面的`localhost:9200/索引名称/类型名称/文档id/_update`方法：

删除文档：

```
DELETE test3/_doc/1    这是删除某一条文档
DELETE test2/          这是删除整个索引
```

文档的操作：

查询：

简单版本：

```
GET yulin/_doc/_search?q=name:裕麟
```

复杂版本：

```
GET yulin/_doc/_search
{
  "query":{                /*查询*/
    "match":{              /*查询机制：匹配查询*/
      "name": "裕麟"       /*匹配项目：name*/
    }
  },
  "_source": "age",        /*显示查询返回结果："age"，相当于MySQL中 select age from
...*/
  "sort": [                /*返回结果排序*/
    {
      "age":{              /*排序依据：age*/
        "order": "asc"     /*排序方式：升序asc，  降序desc*/
      }
    }
  ],
  "from": 0,               /*分页查询开始位置*/
  "size": 1                /*分页查询，每页大小*/
}
```

对于不进行排序的搜索来说，我们在结果中会发现有若干个字段，比如"hits"表示查找到的相关命中项。

```
/*结果:*/
"hits" : {
    "total" : {
      "value" : 3,              /*本次查询总共命中3个*/
      "relation" : "eq"         /*本次查询使用的规则是"eq"相等*/
    },
    "max_score" : 0.6768591,     /*查询最高分*/
    "hits" : [                  /*对hits进行遍历*/
      {
        "_index" : "yulin",      /*当前索引*/
        "_type" : "_doc",        /*文本类型*/
        "_id" : "4",             /*文本id*/
        "_score" : 0.6768591,    /*当前文本搜索得到的分数 --> 分数越高排的越前，这相当于
搜索结果权重*/
        "_source" : {            /*文本字段*/
          "name" : "谢裕麟",      /*name字段*/
          "age" : 4              /*age字段*/
        }
      },
      ……
```

使用term对文本进行搜索的时候，我们可以发现 `text` 类型的字段会被分词解析，而 `keyword` 类型的字段则会作为一个整体进行解析，不会被分词。

集成Springboot

查看官方文档

https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high.html

pom.xml导入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>     <!---->
</dependency>
```

点进去可以看到具体依赖的是：

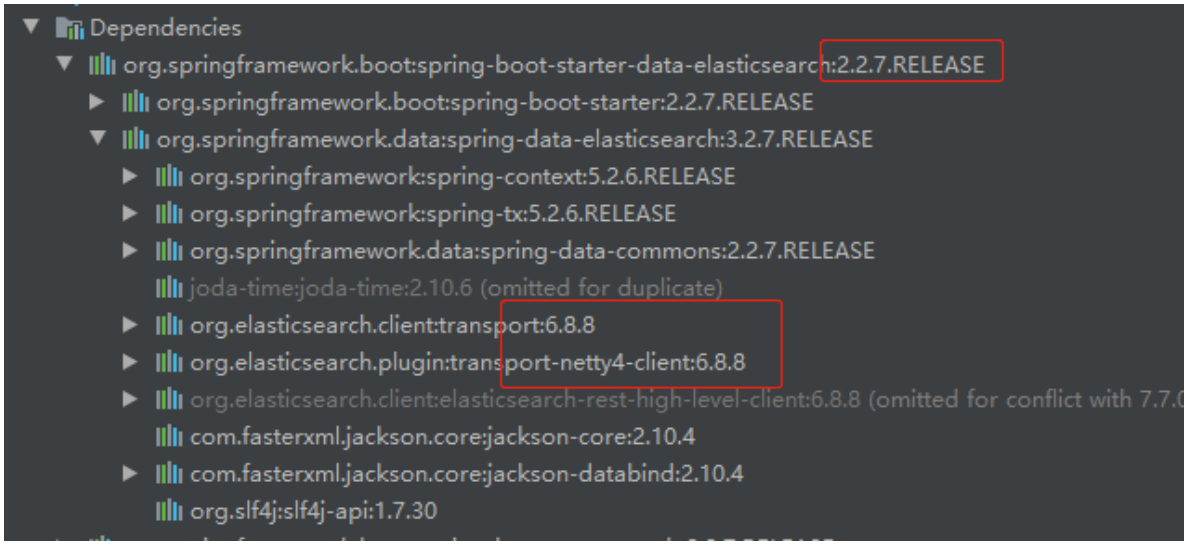
```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-elasticsearch</artifactId>                    <!---->
  <version>3.2.7.RELEASE</version>
  <scope>compile</scope>
  <exclusions>
    <exclusion>
      <artifactId>jcl-over-slf4j</artifactId>
      <groupId>org.slf4j</groupId>
    </exclusion>
    <exclusion>
      <artifactId>log4j-core</artifactId>
      <groupId>org.apache.logging.log4j</groupId>
    </exclusion>
```

```
  </exclusions>
</dependency>
```

再往深一层可以看到依赖的是：

```
<dependency>
   <groupId>io.netty</groupId>                                        <!--netty-->
   <artifactId>netty-bom</artifactId>
   <version>${netty}</version>
   <type>pom</type>
   <scope>import</scope>
</dependency>
<dependency>
    <groupId>org.elasticsearch.client</groupId>                        <!--早期客户
端API-->
    <artifactId>transport</artifactId>
    <version>${elasticsearch}</version>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <!-- required by elasticsearch -->
    <groupId>org.elasticsearch.plugin</groupId>                        <!--插件-->
    <artifactId>transport-netty4-client</artifactId>
    <version>${elasticsearch}</version>
</dependency>
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>        <!--高级
API-->
    <version>${elasticsearch}</version>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

注意到这里有版本适配问题：



因此需要自定义es版本依赖，保证和本地一致。我们在`pom.xml`中的`spring-boot-starter-parent`中的`spring-boot-dependencies`可以找到有`<elasticsearch.version>6.8.8</elasticsearch.version>`版本依赖，只需要在`pom.xml`中添加一句，修改为：

```
<properties>
    <java.version>1.8</java.version>
    <elasticsearch.version>7.6.1</elasticsearch.version>        <!--在这里-->
</properties>
```

配置类

我们在

中可以看到初始化需要一个`RestHighLevelClient`，按照springboot的思想我们需要构造一个Configuration配置类，在里面以返回Bean的方式来将其注入容器中：

```
package yulin.elasticsearch.springboot.config;

import org.apache.http.HttpHost;
import org.elasticsearch.client.RestClient;
import org.elasticsearch.client.RestHighLevelClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * 找对象。放到spring容器中待用。分析源码 xxxAutoConfiguration 和 xxxProperties
 */
@Configuration
public class ElasticSearchConfig {
    @Bean
    public RestHighLevelClient restHighLevelClient(){
        RestHighLevelClient client = new RestHighLevelClient(
                RestClient.builder(
                        new HttpHost("localhost", 9200, "http")));
        return client;
    }
}
```

源码分析

然后我们可以在`package org.springframework.boot.autoconfigure.elasticsearch.rest;`中找到`RestClientAutoConfiguration`这个自动配置类，以及在`package org.springframework.boot.autoconfigure.data.elasticsearch;`中找到`ElasticsearchProperties`。这就是狂神说的分析源码中**xxxAutoConfiguration**和**xxxProperties**文件来获取更多信息。

```
package org.springframework.boot.autoconfigure.elasticsearch.rest;
//...
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RestClient.class)
@EnableConfigurationProperties(RestClientProperties.class)
@Import({ RestClientConfigurations.RestClientBuilderConfiguration.class,
      RestClientConfigurations.RestHighLevelClientConfiguration.class,
      RestClientConfigurations.RestClientFallbackConfiguration.class })
public class RestClientAutoConfiguration {}
```

这里虽然Import导入了3个类，但是都是静态内部类，核心类只有**RestClientConfigurations**这一个。

比如，通过分析上面@Import中的`RestClientConfigurations.RestHighLevelClientConfiguration`类，我们可以发现其中有这么一段：

```
class RestClientConfigurations {
    //...
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(RestHighLevelClient.class)
    static class RestHighLevelClientConfiguration {

       @Bean
       @ConditionalOnMissingBean
       RestHighLevelClient elasticsearchRestHighLevelClient(RestClientBuilder
restClientBuilder) {
          return new RestHighLevelClient(restClientBuilder);
       }
    }
}
```

也就是和我们上面自己配置的ElasticSearchConfig中自定义向Spring容器添加的Bean相同类型。

测试关于索引的API

```
@SpringBootTest
class SpringbootApplicationTests {
    @Autowired
    RestHighLevelClient restHighLevelClient;

    /**
     * 测试索引的创建
     */
    @Test
    void testCreateIndex() throws IOException {
        //创建索引请求
        CreateIndexRequest request = new CreateIndexRequest("test_index");
        //客户端执行请求
```

```
        CreateIndexResponse createIndexResponse =
restHighLevelClient.indices().create(request, RequestOptions.DEFAULT);

        System.out.println(createIndexResponse);
    }

    /**
     * 判断索引是否存在
     */
    @Test
    void testExistIndex() throws IOException {
        GetIndexRequest request = new GetIndexRequest("test_index");
        boolean exists = restHighLevelClient.indices().exists(request,
RequestOptions.DEFAULT);
        System.out.println(exists);
    }

    /**
     * 测试删除索引
     */
    @Test
    void testDeleteIndex() throws IOException {
        DeleteIndexRequest request = new DeleteIndexRequest("test_index");
        AcknowledgedResponse acknowledgedResponse =
restHighLevelClient.indices().delete(request, RequestOptions.DEFAULT);
        System.out.println(acknowledgedResponse.isAcknowledged());
    }
}
```

测试关于文档的API

新建文档:

```
package yulin.elasticsearch.springboot.pojo;
@Component
@AllArgsConstructor
@NoArgsConstructor
@Data
public class User {
    private String name;
    private int age;
}
/*********************************************************************************
*/
package yulin.elasticsearch.springboot;
/**
* 测试增加文档
* @throws IOException
*/
@Test
void testAddDocument() throws IOException {
    //创建新索引请求
    IndexRequest indexRequest = new IndexRequest("test_index");
    //设置索引id : “put test_index/_doc/1” 中的1
    indexRequest.id("1");
    indexRequest.timeout(TimeValue.timeValueSeconds(1));
    //使用阿里巴巴的fastjson将自定义pojo对象转换成json格式，并放入请求中
```

```
    indexRequest.source(JSON.toJSONString(new User("yulin",23)),
XContentType.JSON);
    //获取索引响应
    IndexResponse indexResponse = restHighLevelClient.index(indexRequest,
RequestOptions.DEFAULT);
    //查看响应结果
    System.out.println(indexResponse.toString());
    System.out.println(indexResponse.status());
}
```

会有这样的结果：

```
IndexResponse[index=test_index,type=_doc,id=1,version=1,result=created,seqNo=0,p
rimaryTerm=1,shards={"total":2,"successful":1,"failed":0}]
CREATED
```

修改文档：

```
/**
 * 测试更新文档
 */
@Test
void testUpdateDocument() throws IOException {
    UpdateRequest updateRequest = new UpdateRequest("test_index", "1");
    updateRequest.timeout("1s");

    updateRequest.doc(JSON.toJSONString(new User("yulin!!", 33)),
XContentType.JSON);
    UpdateResponse updateResponse = restHighLevelClient.update(updateRequest,
RequestOptions.DEFAULT);

    System.out.println(updateResponse.toString());
    System.out.println(updateResponse.status());
}
```

结果如下，可以对比上面创建时的返回值：

```
UpdateResponse[index=test_index,type=_doc,id=1,version=2,seqNo=1,primaryTerm=1,r
esult=updated,shards=ShardInfo{total=2, successful=1, failures=[]}]
OK
```

查找文档：

```
/**
 * 测试查找文档
 */
@Test
void testGetDocument() throws IOException {
    GetRequest getRequest = new GetRequest("test_index", "1");
    GetResponse getResponse = restHighLevelClient.get(getRequest,
RequestOptions.DEFAULT);

    System.out.println(getResponse.toString());
    System.out.println(getResponse.getIndex());
    System.out.println(getResponse.getSource());
}
```

结果如下，可以看到基本上想要的信息都可以通过 `getResponse.getXXX()` 函数获得：

```
{"_index":"test_index","_type":"_doc","_id":"1","_version":2,"_seq_no":1,"_prima
ry_term":1,"found":true,"_source":{"age":33,"name":"yulin!!"}}
test_index
{name=yulin!!, age=33}
```

删除文档：

```
/**
 * 测试删除文档
 */
@Test
void testDeleteDocument() throws IOException {
    DeleteRequest deleteRequest = new DeleteRequest("test_index", "1");
    deleteRequest.timeout("1s");

    DeleteResponse deleteResponse = restHighLevelClient.delete(deleteRequest,
RequestOptions.DEFAULT);
    System.out.println(deleteResponse.toString());
    System.out.println(deleteResponse.status());
}
```

结果如下：

```
DeleteResponse[index=test_index,type=_doc,id=1,version=3,result=deleted,shards=S
hardInfo{total=2, successful=1, failures=[]}]
OK
```

批量操作：

```
/**
 * 批量插入数据
 */
@Test
void testBulkRequest() throws IOException {
    BulkRequest bulkRequest = new BulkRequest();
    bulkRequest.timeout("1s");

    ArrayList<User> list = new ArrayList<>();
```

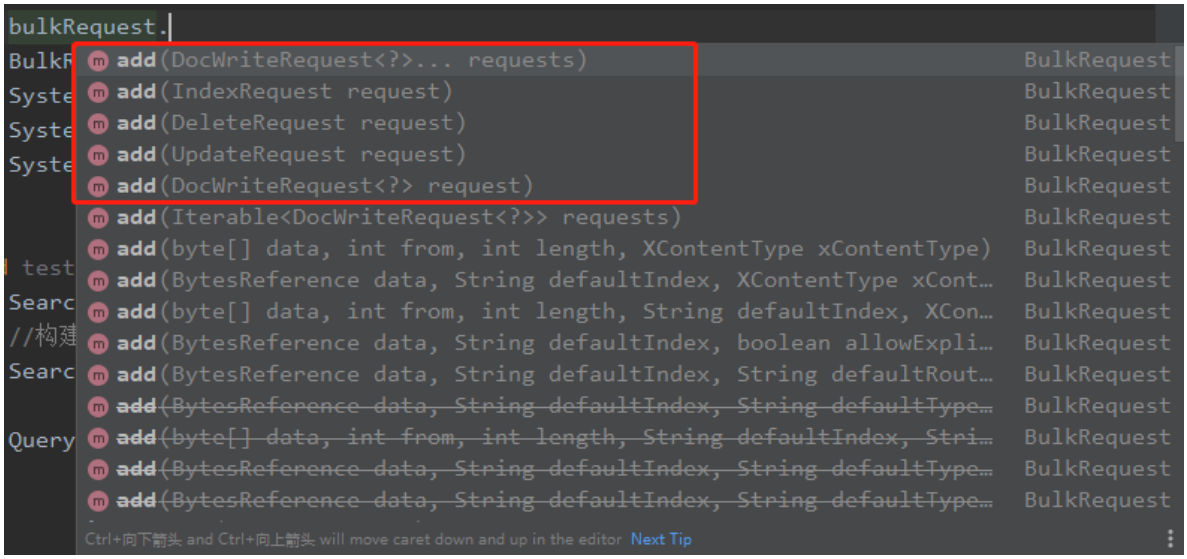
```
    list.add(new User("yulin1",1));
    list.add(new User("yulin2",2));
    list.add(new User("yulin3",3));
    list.add(new User("yulin4", 4));

    //这里是批量插入，我们也可以使用批量删除修改查找等，只需要找重载的add方法
    for (int i = 0; i < list.size(); i++) {
        bulkRequest.add(
                new IndexRequest("test_index").id(""+(i+1))

.source(JSON.toJSONString(list.get(i)),XContentType.JSON));
    }

    BulkResponse bulkResponse = restHighLevelClient.bulk(bulkRequest,
RequestOptions.DEFAULT);
    System.out.println(bulkResponse.toString());
    System.out.println(bulkResponse.hasFailures());
    System.out.println(bulkResponse.status());
}
```

上面这里是批量插入，我们也可以使用批量删除修改查找等，只需要找重载的add方法：



搜索Search：

```
@Test
void testSearch() throws IOException {
    SearchRequest searchRequest = new SearchRequest("test_index");
    //构建搜索条件
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
    //查询条件，我们可以使用QueryBuilders工具类来实现
    //MatchAllQueryBuilder matchAllQueryBuilder = QueryBuilders.matchAllQuery();
    //TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery();
    TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("name",
"yulin1");
    searchSourceBuilder.query(termQueryBuilder);
    searchSourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));

    searchRequest.source(searchSourceBuilder);

    SearchResponse searchResponse = restHighLevelClient.search(searchRequest,
RequestOptions.DEFAULT);
```

```
    System.out.println(searchResponse.getHits());
    System.out.println(Arrays.toString(searchResponse.getHits().getHits()));
}
```